Soutěž dětí a mládeže v programování

Obvodní kolo Prahy 1 a 2, rok 2026

Časté chyby, zajímavosti řešení

V řešeních kraluje Python. Druhý nejčastější je C#. Další jazyky (C, C++, webové technologie jako je Svelte) jsou výjimečné.

Úloha 1: Slovní fotbal - hra

Zadání mluví o podpoře diakritky - myslí se tím, že když slovo končí na “ň”, další by mělo na “ň” začínat - nikoliv na “n”.
`unicodedata.normalize('NFKD', word)` je tedy pro naše účely chybou - ale pro některá jiná použití se vám může podobná normalizace diakritiky do základních ascii znaků hodit - například při interakci s nečím co háčky a čárky nepodporuje (doménové jméno, soubory na disku…).

Jak řešit ch? Vzhledem k tomu že je to naše jediná spřežka, je v pořádku ji věnovat separátní větev:

```
def posledni_pismeno(slovo):
    if slovo.endswith('ch'):
        return 'ch'
    else:
        return slovo[-1]
```

Další důležitý ohled hodnocení bylo, zda hra umožňuje hráči vzdát se (napsat hráči, ať stiskne “Ctrl+C pro vzdání se” je minimalistické řešení, které mě pobavilo - lepší bylo mít nějaký specifický řetězec, např. “konec” a nebo za to považovat prázdný řetězec)

Úloha 2: Slovní fotbal - nejdelší řetěz

Jde vlastně o grafovou úlohu.

> *Než půjdeme dál: „graf" v informatice neznamená graf funkce jako ve škole. Jde o síť „vrcholů" (bodů) spojených „hranami" (čarami). Představte si třeba mapu měst a silnic mezi nimi.*

V našem případě budou vrcholy grafu jednotlivá písmena abecedy (těch je i s diakritikou 42) a hranami slova - slovo v nějakém vrcholu (písmenku) začíná a v nějakém končí.

Mimochodem to znamená, že graf je orientovaný (slova směřují) a že může být více hran mezi dvěma vrcholy, například slova “kobra” a “kráva” vedou obě z K do A

Jde o rozhodnutí, zda v tom grafu existuje Eulerovský tah, neboli tah který projde správným směrem všemi hranami (tj. použije všechna slova), aniž by nějakou vynechal nebo opakoval.

Fascinující je, že pro rozhodnutí, zda je takový tah možný, stačí spočítat kolik hran začíná v jednotlivých vrcholech (tedy kolik slov začíná na které písmeno) a kolik hran končí v jednotlivých vrcholech (tedy kolik slov končí na které písmeno).

- **Uzavřený tah** (začíná i končí ve stejném slově): každé písmeno musí mít stejný počet slov, která na něj začínají, i slov, která na něj končí = ANO
- **Otevřený tah**: přesně jedno písmeno má o jedno „začínající slovo" navíc (startovní písmeno) a přesně jedno má o jedno „koncové slovo" navíc. Vše ostatní musí být vyrovnané = ANO
- Jakýkoli jiný nepoměr = NE, řetěz všech slov neexistuje.

Rozhodnutí ANO/NE je tedy možné učinit efektivně, pouze spočtením začátků a konců slov (v Pythonu například ve struktuře Counter)

Vytvoření nejdelšího tahu (tedy řetězu slov) je potom úkol pro Hierholzerův algoritmus. Začněte od startovního slova (nebo libovolného, pokud je tah uzavřený) a jděte dál, dokud to jde. Výsledná cesta nemusí zahrnovat všechna slova - proto procházejte slova na cestě a z každého, kde ještě zbývají nevyužité hrany, přidávejte další kružnice (řetězy slov začínající i končící na stejné písmeno), dokud nejsou využity všechny hrany.

Nejčastější chybou bylo použití hladového algoritmu z první úlohy - jednoduché “nechte hrát dva počítače proti sobě”. Takové řešení sice zvládne jednoduchý vstup (jde01), ale pro složitější vstupy nenajde správnou cestu nebo vůbec neskončí.

Druhá opakující se chyba byla chybějící podpora ch - slova jako “chochol" nebo “moloch” pak program zpracuje špatně. Stačí jedna pomocná funkce (viz úloha 1).

Třetí chybou byla NFKD normalizace diakritiky, kvůli které program špatně určoval počáteční a koncová písmenka (patrné ve jde04.json).

Několik řešení trpělo problémy s rekurzí - Python má výchozí limit hloubky zásobníku kolem 1000 volání, což pro velké slovníky nestačí. Hierholzerův algoritmus je přitom přirozeně iterativní a rekurzi nepotřebuje.

Úloha 3: Defragmentace

Tato úloha nemá žádný zásadní chyták. Spočívá v provádění pouze tolik operací kolik je potřeba a ne více.

Některé implementace používali regex pro hledání odkazů na ostatní soubory. Regex ale může být občas problematický a je to zbytečně velké kladivo na hledání dvou znaků “[” “]”.

Triviální řešení pouze rekurzivně prochází soubory a kopíruje obsah. Není nutné každý soubor zpracovávat zvlášť. Naopak když provádíme pouze jeden průchod celým výsledkem, tak budeme kopírovat data nejméně.

Pokud budeme načítat celý vstup do paměti a poté v něm nahrazovat odkazy, tak budeme velké množství textu přesouvat sem a tam. Nahrazování prostředku textu větším obsahem není triviální ačkoliv to v mnoha programovacích jazycích tak může vypadat. Teoreticky by šlo nějak optimalizovaně poskládat výsledný text chytře v paměti za pomoci provazů a copy-on-write nebo cache s nevlastněnými řetězci. Naše zadání je ale vytvořit defragmentovaný soubor, a pro takový účel je triviální řešení velmi blízko optimálnímu. Pokud vás zajímá více o zmíněných konceptech - provaz, copy-on-write.

Při procházení souborů můžeme narazit na smyčku. Pro správnou detekci smyčky ale potřebujeme určit jestli někdy procházíme soubor, který už jsme prošli. Jak zní zadání, tak soubory na sebe mohou odkazovat napříč různými adresáři. Takže pro rozlišení potřebujeme celou cestu.

Zároveň ale také musíme vzít v potaz skládání cest. Cesty “/lorem/ipsum/../dolor/sit.txt” a “/lorem/dolor/sit.txt” video ke stejnému souboru. Jednoduchý způsob jak sjednotit tyto cesty je kanonikalizace (hledejte jako canonicalize). Požádáme OS o zjednodušení cesty k souboru na minimální. To ale může být pomané, protože provádíme systémová volání. Druhá možnost je normalizace (“normalize” nebo “lexically normalize”), která pouze triviálně smaže všechny “/./” a pro každý “/../” odebere nadřazenou složku. Normalizace lze provést bez volání do systému a může tedy být o trochu rychlejší. Obě možnosti běžně bývají ve standardních knihovnách.

Typická optimalizace této defragmentace je cachování souborů. Pokud se nějaký soubor vyskytuje ve výsledku mnohokrát, tak bychom ho museli pokaždé načíst znovu. Abychom zpracovali soubor, musíme ho načíst do paměti. Takže nezavedeme moc práce navíc tím že si budeme obsahy souborů ukládat do paměti, abychom je nemuseli načítat opakovaně.

Úloha 4: Analogové hodiny